EAST Search History

Ref #	Hits	Search Query	DBs	Default Operator	Plurals	Time Stamp
L1	38827	"707"/$.ccls.	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	ON	2006/11/12 18:11
L2	49356	"709"/$.ccls.	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	ON	2006/11/12 18:11
L3	27378	"715"/$.ccls.	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	ON	2006/11/12 18:11
L4	105907	1 or 2 or 3	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	ON	2006/11/12 18:19
L5	14310	((document$1 with conversion) or (convert$3 near document$1))	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	ON	2006/11/12 18:21
L6	3150	((document$1 with conversion) or (convert$3 near document$1)) and ((structured with document$1) or SGML or XML or HTML)	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	ON	2006/11/12 18:22
L7	2711	((document$1 with conversion) or (convert$3 near document$1)) and ((structured with document$1) or SGML or XML or HTML) and (transform or transformation or format$5 or "XSL Transformation")	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	ON	2006/11/12 18:24
L8	1561	(((document$1 with conversion) or (convert$3 near document$1)) and ((structured with document$1) or SGML or XML or HTML) and (transform or transformation or format$5 or "XSL Transformation")) and 4	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	ON	2006/11/12 18:24

EAST Search History

L9	4	((structur$4 with pattern$1) same (((document$1 with conversion) or (convert$3 near document$1)) and ((structured with document$1) or SGML or XML or HTML) and (transform or transformation or format$5 or "XSL Transformation"))) and 4	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	ON	2006/11/12 18:29
L10	19	((re-arrange$4 or rearrang$4 or arrang$4) same (((document$1 with conversion) or (convert$3 near document$1)) and ((structured with document$1) or SGML or XML or HTML) and (transform or transformation or format$5 or "XSL Transformation"))) and 4	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	ON	2006/11/12 18:30
L11	0	(((re-arrange$4 or rearrang$4 or arrang$4) same (((document$1 with conversion) or (convert$3 near document$1)) and ((structured with document$1) or SGML or XML or HTML) and (transform or transformation or format$5 or "XSL Transformation"))) and 4) and (candidate with list$1)	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	ON	2006/11/12 18:30

	Document ID	Kind Codes	Source	Issue Date	Pages
1	US 20060168095 A1		US-PGPUB	20060727	51
2	US 20060161840 A1		US-PGPUB	20060720	12
3	US 20060085739 A1		US-PGPUB	20060420	16
4	US 20060041840 A1		US-PGPUB	20060223	104
5	US 20050193009 A1		US-PGPUB	20050901	13
6	US 20050160109 A1		US-PGPUB	20050721	55
7	US 20050066271 A1		US-PGPUB	20050324	38
8	US 20040168125 A1		US-PGPUB	20040826	11
9	US 20040103075 A1		US-PGPUB	20040527	21
10	US 20040044963 A1		US-PGPUB	20040304	42
11	US 20030145062 A1		US-PGPUB	20030731	37
12	US 20010056418 A1		US-PGPUB	20011227	21
13	US 20010014900 A1		US-PGPUB	20010816	19

	Title	Abstract
1	Multi-modal information delivery system	
2	Methodology for mapping HL7 V2 standards to HL7 V3 standards	
3	Document processing apparatus and control method thereof	
4	File translation methods, systems, and apparatuses for extended commerce	
5	System and method for hierarchical data document modification	
6	Apparatus and method for managing date files	
7	Extraction of information from structured documents	
8	Method and system for generating and finishing documents	
9	International information search and delivery system providing search results personalized to a particular natural language	
10	Extraction of information from structured documents	
11	Data conversion server for voice browsing system	
12	System and method for facilitating internet search by providing web document layout image	
13	Method and system for separating content and layout of formatted objects	

	Current OR	Current XRef	Retrieval Classif	Inventor
1	709/217			Sharma; Dipanshu et al.
2	715/513	705/3; 715/523		Cohen; Simona et al.
3	715/513			Iwasaki; Shingo
4	715/513	715/523		Blair; William R. et al.
5	707/104.1			Reinhardt, Tilo et al.
6	707/101			Sato, Hisashi et al.
7	715/513			Uchiyama, Tadasu et al.
8	715/523			van der Meer, Hendrik Theodorus et al.
9	707/1			Kim, Moom Ju et al.
10	715/513			Uchiyama, Tadasu et al.
11	709/217			Sharma, Dipanshu et al.
12	707/3	707/10; 707/104.1		Youn, Seok Ho
13	715/513	715/517		Brauer, Michael et al.

	Document ID	Kind Codes	Source	Issue Date	Pages
14	US 7099861 B2		USPAT	20060829	21
15	US 7092938 B2		USPAT	20060815	13
16	US 7039625 B2		USPAT	20060502	21
17	US 6564256 B1		USPAT	20030513	9
18	US 6061697 A		USPAT	20000509	52
19	US 5694609 A		USPAT	19971202	46

	Title	Abstract
14	System and method for facilitating internet search by providing web document layout image	
15	Universal search management over one or more networks	
16	International information search and delivery system providing search results personalized to a particular natural language	
17	Image transfer system	
18	SGML type document managing apparatus and managing method	
19	Document processing apparatus for processing a structured document using embedding nodes and mold nodes	

	Current OR	Current XRef	Retrieval Classif	Inventor
14	707/3	707/6		Youn; Seok Ho
15	707/4	707/3		Brown; Gregory T. et al.
16	707/1	704/2; 707/100		Kim; Moom Ju et al.
17	709/219	709/203; 719/328		Tanaka; Nobuyuki
18	715/513	707/203; 707/3; 715/517		Nakao; Yoshio
19	715/513			Murata; Makoto

Results for "((structur* and pattern* and document* and conversion)<in>metadata)"

e-mail printer

Your search matched 4 of 1431298 documents.

A maximum of 100 results are displayed, 25 to a page, sorted by Relevance in Descending order.

» Search Options

View Session History

New Search

Modify Search

((structur* and pattern* and document* and conversion)<in>metadata) Search

Check to search only within this results set

» Key

IEEE JNL	IEEE Journal or Magazine
IEE JNL	IEE Journal or Magazine
IEEE CNF	IEEE Conference Proceeding
IEE CNF	IEE Conference Proceeding
IEEE STD	IEEE Standard

Display Format: Citation Citation & Abstract

view selected items Select All Deselect All

1. **Structured document framework for design patterns based on SGML**
Ohtsuki, M.; Segawa, J.; Yoshida, N.; Makinouchi, A.;
Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annua International
13-15 Aug. 1997 Page(s):320 - 323
Digital Object Identifier 10.1109/CMPSAC.1997.624848
AbstractPlus | Full Text: PDF(508 KB) IEEE CNF
Rights and Permissions

2. **Document image processing based on enhanced border following algorithm**
Yamada, M.; Hasuike, K.;
Pattern Recognition, 1990. Proceedings., 10th International Conference on
Volume ii, 16-21 June 1990 Page(s):231 - 236 vol.2
Digital Object Identifier 10.1109/ICPR.1990.119360
AbstractPlus | Full Text: PDF(484 KB) IEEE CNF
Rights and Permissions

3. **Analysis and conversion of documents**
Tayeb-Bey, S.; Saidi, A.S.; Emptoz, H.;
Pattern Recognition, 1998. Proceedings. Fourteenth International Conference on
Volume 2, 16-20 Aug. 1998 Page(s):1089 - 1091 vol.2
Digital Object Identifier 10.1109/ICPR.1998.711882
AbstractPlus | Full Text: PDF(108 KB) IEEE CNF
Rights and Permissions

4. **Linear layout processing**
Ting, A.; Leung, M.K.H.;
Pattern Recognition, 1998. Proceedings. Fourteenth International Conference on
Volume 1, 16-20 Aug. 1998 Page(s):403 - 405 vol.1
Digital Object Identifier 10.1109/ICPR.1998.711166
AbstractPlus | Full Text: PDF(52 KB) IEEE CNF
Rights and Permissions

Results for "((structural and document<near/2>conversion)<in>metadata)"

Your search matched 2 of 1431298 documents.

A maximum of 100 results are displayed, 25 to a page, sorted by Relevance in Descending order.

1. **Automatic understanding of structures in printed mathematical expressions**
   Mitra, J.; Garain, U.; Chaudhuri, B.B.; Kumar Swamy HV; Pal, T.;
   Document Analysis and Recognition, 2003. Proceedings. Seventh International Conference on
   3-6 Aug. 2003 Page(s):540 - 544 vol.1
   Digital Object Identifier 10.1109/ICDAR.2003.1227723

2. **Design of documentation mediation system for mobile service**
   Sunghan Kim; Changsu Kim; YoungTae Park; Kishik Park; Hoekyung Jung; YoungTae Kim; HoeKyung Jung;
   Computer Networks and Mobile Computing, 2001. Proceedings. 2001 International Conference on
   16-19 Oct. 2001 Page(s):189 - 193
   Digital Object Identifier 10.1109/ICCNMC.2001.962595

Web Results **1** - **10** of about **12** for **structure document convert conversionpattern rearranging**. (**0.42** seconds)

Rational® XDE(tm) Release Notes
It is important to review this **document** because it describes: ... (examples: project and model creation, Rose model **conversion, Pattern** application). ...
it-des-group.web.cern.ch/it-des-group/ETS/swdevtools/sdt/XDE/release_notes.html - 223k - Cached - Similar pages

Imaging device and a monitoring system - Patent 20040179100
[0074] The A/D **converter** 204 converts the analog image signals of R, ... [0095] In order to solve such a problem, the image **rearranging** unit 2071 performs a ...
www.freepatentsonline.com/20040179100.html - 96k - Cached - Similar pages

> Automatic accompaniment device capable of selecting a desired ...
> Next, the **structure** of the personal computer 20 will be described in detail. ... data base means and **rearrange** the order of the data within pattern table 63 ...
> www.freepatentsonline.com/5756917.html - 205k - Cached - Similar pages

[PS] postscript
File Format: Adobe PostScript - View as HTML
Each **conversion pattern** begins with a percent symbol (%) and is followed by ... **Convert** this set to an Entry data **structure** and print the description and ...
www.inf.ethz.ch/personal/cgina/Book/drive.ps - Similar pages

[PDF] MANAGING DOMAIN ARCHITECTURE EVOLUTION THROUGH ADAPTIVE USE CASE ...
File Format: PDF/Adobe Acrobat
This business rule pattern is identical to the Type **Conversion pattern** except ... can **document** design. Moreover, these statements match the **structure** of the ...
www.iit.edu/~rhurlbut/hurl98.pdf - Similar pages

[PDF] Calyxo Intro
File Format: PDF/Adobe Acrobat - View as HTML
The **converter** takes a string as input and produces an object as result. ... Optionally, **rearrange** resource locations to reflect the new module **structure**. ...
calyxo.org/calyxo-root.pdf - Similar pages

> [PDF] Calyxo MVC Web Application Framework
> File Format: PDF/Adobe Acrobat
> The **converter** takes a string as input and produces an object as result. ... Thus, our configuration **document** has a general **structure** like this: ...
> calyxo.org/calyxo.pdf - Similar pages

[PDF] Struts Survival Guide
File Format: PDF/Adobe Acrobat
A clear understanding of the **structure** of Struts web applications is key to ... methods on it and **convert** the implementation exceptions into application ...
www.objectsource.com/Struts_Survival_Guide.pdf - Similar pages

[PDF] System Administration Guide
File Format: PDF/Adobe Acrobat
directory **structure** and backup procedures for Enterprise Integrator. ... Process in which converters specified in message definitions **convert** an ...
www.iona.com/support/docs/enterprise_integrator/5.6/ei_admin_guide/ei_admin_guide.pdf - Similar pages

SolarSoft Library

(This differs from the IDL SORT routine alone, which may **rearrange** order for ... header info
Input Parameters: doc_str - **document structures**, as returned by ...
www.astro.washington.edu/deutsch/idl/htmlhelp/library32.html - Similar pages

Result Page: 1 2 **Next**

Up-conversion using XSLT 2.0

Keywords: XSLT

Michael Kay
Director
Saxonica Limited
Reading
United Kingdom
mike@saxonica.com

Biography

Michael Kay is the editor of the XSLT 2.0 specification, the developer of the Saxon XSLT and XQuery processor, and the author of the books *XSLT 2.0 Programmer's Reference* and *XPath 2.0 Programmer's Reference* from Wrox Press.

He founded Saxonica Limited earlier this year to continue the development of the Saxon software and to provide consultancy services to XSLT and XQuery users.

Abstract

XSLT 2.0 provides a wide range of new features, many of which make light work of tasks that are notoriously difficult in XSLT 1.0, such as grouping and string manipulation. This paper attempts to show how these facilities not only make coding easier, but will also extend the scope of the language making it possible to tackle problems that were quite outside the range of XSLT 1.0.

The paper shows case study of a multi-phase transformation taking data from a legacy ASCII-based interchange format, to XML based on a standardized vocabulary. The transformations illustrate the power of new features including regular expression handling, grouping, recursive functions, and schema-aware processing.

The conclusion of the paper is that these new facilities - notably regular expression handling and grouping - take XSLT into new territory, where languages such as Perl previously reigned supreme. XSLT 1.0 works best where all the structure in a document is already identified by markup. XSLT 2.0 will also be able to handle many situations where the structure is implicit in the text, or in markup designed for presentation purposes rather than to capture the information semantics. It thus becomes a powerful tool for "up-conversion" applications. These facilities

work will in conjunction with schema-aware processing, where the aim of the exercise is to create XML that conforms to a target schema.

Table of Contents

1. Introduction

XSLT 1.0 became a W3C Recommendation in November 1999; it has attracted at least twenty implementations and a very sizeable user base. It is used mainly for two distinct applications: rendering of XML documents by converting them into a presentation-oriented vocabulary (usually HTML, sometimes XSL-FO, XHTML, or SVG); and conversion of data-oriented XML messages, either into a different vocabulary, or to a different document using the same vocabulary, but with different information content. Within these two categories there are some highly creative and innovative applications, a notable example being Schematron, which uses XSLT transformations to apply structural and semantic validation rules to a document.

Although XSLT 1.0 is designed to transform source XML trees into result XML trees, it also includes three serialization methods, allowing the result tree to be output either as lexical XML, HTML, or text. This enables a wide range of applications in which the output is in textual form: I have seen XSLT stylesheets that generated Java programs, SQL code, comma-separated-values files, and EDI messages.

However, this ability to generate multiple output formats is not mirrored on the input side. XSLT 1.0 has very little capability to take anything other than XML as its input. There are ways around this: for example

in the first edition of my book *XSLT Programmer's Reference* I showed how one could write a parser for a non-XML format such as the GEDCOM 5.5 format used for genealogical data, and by making this parser implement the SAX interface supported by many XSLT processors, one could present the parsed input data to the XSLT 1.0 processor as if it came from an XML parser. However, this is really only a minor improvement on what can be achieved by writing a GEDCOM-to-XML converter as a standalone application.

XSLT 2.0, as I will show in this paper, greatly extends the ability of XSLT to process any textual input, without the need to write conversion code in Java or another procedural programming language. It therefore enables XSLT to be used not only for XML-to-XML and XML-to-text applications, but also for text-to-XML conversions. More generally, it allows XSLT 2.0 to be used for up-conversion.

In the broadcasting industry, the term *upconversion* (usually without a hyphen) is used to mean the conversion of a low-frequency video format to an equivalent high-frequency format. In the SGML and XML world, the word refers to the generation of a format with detailed markup from a format with less-detailed or no markup, where it is necessary to generate the additional markup by recognizing structural patterns that are implicit in the textual content itself. By extension the term is also used for converting non-SGML or non-XML markup into SGML/XML: this usage is justified, of course, on the basis that SGML/XML is obviously on a higher plane than any alternative markup language!

I will start this paper with a survey of the new features in XSLT 2.0 that make it easier to write up-conversion transforms (it really doesn't make much sense to call them stylesheets any more, but I will slip into that usage occasionally). I will then present a case study of a particular up-conversion. I will use the example I mentioned earlier, conversion of GEDCOM genealogical data: but this time, the entire job will be done in XSLT 2.0, with no need to write preprocessing software in a procedural language.

2. Up-Conversion Facilities in XSLT 2.0

In this section I will describe how four of the new features in XSLT 2.0 can be used to assist in writing up-conversion applications. The four features discussed are:

- The unparsed-text() function
- Regular expression processing
- Grouping facilities
- Schema-aware processing

The descriptions here are brief introductions to these facilities: for full information, see the W3C specifications of XSLT 2.0 **[XSLT 2.0]** and XPath 2.0 **[XPath 2.0]**, or my books *XSLT 2.0 Programmer's Reference***[Kay, 2004a]** and *XPath 2.0 Programmer's Reference***[Kay, 2004b]**.

2.1 The unparsed-text() function

In order to handle non-XML input, the first thing a stylesheet needs to be able to do is to read it. For this purpose, XSLT 2.0 provides the unparsed-text() function. This takes a URI as its first argument, and loads the text of the resource found at that URI. The result is a character string - that is, a value of type `xs:string`, where "xs" is the XML Schema namespace. The type system of XSLT 2.0 is based on the types defined in the XML Schema specification.

In fact, it was already possible in XSLT 1.0 to provide a stylesheet with non-XML input, in the form of a string-valued stylesheet parameter (parameters can be declared using a global <xsl:param> element). However, this imposes constraints, for example it is difficult to handle a variable number of such inputs. Allowing URI-addressible resources to be accessed directly makes the job much easier.

Character encoding is of course a problem. The unparsed-text() function allows a second parameter to specify the character encoding explicitly, or it can be guessed from external information - the XSLT 2.0 spec refers to the algorithms and heuristics defined in the XLink specification for this purpose. In practice, if the file is an arbitrary file in operating system filestore with no associated metadata, guessing its encoding is sometimes going to give wrong answers. Sadly, there is no easy solution to this difficulty.

The fact that the result of the unparsed-text() function must be an `xs:string` imposes a constraint: the only characters allowed in the file are those permitted in XML documents. This same constraint also applies to any text output produced by a stylesheet. It means that XSLT is now capable of reading textual input and writing textual output, but it cannot be used to handle binary input or binary output, unless these are first translated into some textual representation.

2.2 Regular expression processing

XSLT 1.0 has been much criticized for its rather primitive text-handling capabilities: the function library provided for string handling in XPath 1.0 is designed very much on "reduced instruction set computing" principles - you can achieve pretty well anything, but the complexity of the programming needed even for some quite simple tasks can be daunting. In particular, for many users (whether or not they have a programming background), writing string manipulation routines in terms of recursive templates can present a big conceptual barrier.

I don't know the history of the decisions that brought this situation about. I have always thought the statement at the start of the XSLT 1.0 specification, to the effect that XSLT is not a general-purpose programming language, was very suggestive: committees don't put a statement like that in a specification unless there has been a vigorous debate on the matter, and the fact that the statement is there means there must have been a strong "keep it simple" camp on the working group who won the debate. Which is probably a good thing, given the length of time the world has been waiting for an XQuery recommendation.

But the fact is, there is a large class of applications for which the text processing capability in XSLT 1.0 is woefully inadequate - and this includes most up-conversion applications. XSLT 1.0 is very good at performing structural transformations - that is, at rearranging the nodes in a tree. It is much less good at manipulating the textual content of those nodes. By definition, up-conversion applications are those where

the input doesn't have explicit structure, but rather has structure that is implicit in the text, and therefore they need good text processing capability.

Users of Perl and similar languages have long been accustomed to the power of regular expressions (regexes). In fact, they are so powerful they can become addictive: whereas programmers from other disciplines might turn to regular expressions as a last resort, there are Perl programmers who see almost any problem as an opportunity for creativity in their use of regexes.

XPath 2.0 offers three functions in its standard function library that perform regular expression processing. Specifically:

- **matches()**: returns a boolean value indicating whether a particular string matches a regular expression.
- **replace()**: replaces those substrings within a given string that match a regular expression, with a replacement string.
- **tokenize()**: breaks a string into a sequence of substrings, based on finding delimiters or separators that match a given regular expression.

Conspicuously missing from this list is any function that allows markup to be inserted into a string. It can be done somewhat laboriously by combining the different functions together, but using these three functions alone to translate `See [2]` into `See <ref>2</ref>` is painfully hard work. The reason for the omission is that it's hard to solve the requirement with a simple function.

The XSLT/XQuery/XPath programming model, despite the fact that it owes a great deal to functional programming theory, does not support higher-order functions. That is, functions are not first-class objects and cannot be supplied as arguments to other functions. This greatly limits the power of what can be achieved with a function library alone. All higher-order capabilities in the three languages are instead achieved by means of higher-order operators, custom syntax, or XSLT instructions. An example is the XPath `for` expression, which in a pure functional language would be expressed as a higher-order `map` or `apply` operator taking a sequence as its first argument and a function (to be applied to each member of the sequence) as its second argument; another example is the construct `SEQ[P]` which is essentially a higher-order `filter` function that takes a sequence as its first argument and a predicate as its second.

So the XSLT solution to this problem is an instruction, `xsl:analyze-string`, that logically takes four arguments: the string to be analyzed, a regex, an instruction to be executed to process substrings that match the regular expression, and an instruction to be executed to process substrings that don't match. The earlier example that turns `See [2]` into `See <ref>2</ref>` can then be coded as follows:

```
<xsl:analyze-string select="$input" regex="\[.*?\]">
  <xsl:matching-substring>
    <ref><xsl:value-of select="translate(.,'[]', '')"/></ref>
  </xsl:matching-substring>
  <xsl:matching-substring>
    <xsl:value-of select="."/>
  </xsl:matching-substring>
</xsl:analyze-string>
```

Those who are comfortable with regular expressions will have little difficulty following what `regex="\[.*?\]"` does: `\[` matches an opening square bracket, `.*` matches any sequence of characters, the `?` is a modifier indicating that the `.*` should match the shortest possible sequence of characters consistent with the regex as a whole succeeding, and the `\]` matches a closing square bracket.

The semantics of `xsl:analyze-string` are that the input string is scanned from left to right looking for substrings that match the regex. Substrings that don't match the regex are passed (as the context item, ".") to the `xsl:non-matching-substring` instruction, which in this case copies them unchanged, while substrings that do match the regex are passed to `xsl:matching-substring`, which in this example wraps the substring in a `ref` element, using the (XSLT 1.0) translate() function to drop the delimiting square brackets. (Regex devotees will find a different way of doing this, but the old translate() function suits me fine.)

There is no equivalent facility to `xsl:analyze-string` in XQuery. In the latest release (version 8.1) of Saxon I have introduced an extension to support higher-order functions, and have used this to provide an extension function `saxon:analyze-string`**[Saxonica, 2004]** that takes as its arguments the string to be processed, the regex, and two functions to be applied to the matching and non-matching substrings respectively. It's not quite as convenient to use as the XSLT 2.0 construct, but it demonstrates that if higher-order functions were available in the language, there would be a lot less need for custom syntax to solve such problems.

2.3 Grouping facilities

Grouping problems probably form the largest category of tricky-to-solve problems faced by XSLT 1.0 users. I classify any problem as a grouping problem if it requires the addition of an extra layer of hierarchy in the result tree that is not present in the source tree. Grouping problems fall essentially into two categories: those that group elements having matching data values, and those that group elements based on their position in a sequence (for example, a `heading` element followed by all the `para` elements up to the next `heading`).

XSLT 1.0 offers no inbuilt support for solving grouping problems, and neither does XQuery 1.0. The standard solution for value-based grouping in XSLT 1.0 is a technique using keys, which was invented by Steve Muench of Oracle and is therefore known as Muenchian grouping: its best description is that by Jeni Tennison at **[Tennison]**. (Steve never published it himself: he first described it in a personal email to me, and I announced his discovery to the world. I am very pleased that he got the credit he deserved, which is unusual in our industry.) For positional grouping, a number of techniques are possible, generally involving recursive processing using the following-sibling axis. (Unfortunately neither keys nor the following-sibling axis are available in XQuery, so XQuery users are going to struggle with this one.)

XSLT 2.0 offers a new instruction, `xsl:for-each-group`, to perform grouping. It provides four ways to define the grouping criterion: simple value-based grouping (the most common requirement) can be achieved by defining an expression to compute the grouping key, while the other three options define various kinds of positional grouping criteria. The body of the `xsl:for-each-group` instruction is then executed once for each group of nodes identified.

To take a simple example, the following code takes a flat list of `author` elements, and groups them so that authors with the same affiliation appear as children of an `affiliation` element:

```
<xsl:for-each-group select="author" group-by="affiliation">
  <affiliation name="{current-grouping-key()}">
    <xsl:copy-of select="current-group()"/>
  </affiliation>
</xsl:for-each-group>
```

What is the relevance of this to up-conversion, the subject of this paper? The answer is that up-conversion involves detection of implicit structure, and replacement of the implicit structure by explicit markup. This is exactly what grouping facilities are doing. This time, the implicit structure is not found by parsing the text, but by looking for patterns in the existing markup. This will become very clear in my case study, presented in the second half of this paper.

Like `xsl:analyze-string`, the `xsl:for-each-group` instruction is essentially syntactic sugar for a higher-order function. This time you can think of it (specifically the variant for value-based grouping) as a function whose arguments are the sequence to be grouped, a function to calculate the grouping key, and a function to be evaluated once for each group of items in the input sequence. So that XQuery users can take advantage of the grouping facilities in Saxon, I have again provided a higher-order extension function in Saxon 8.1 that provides this capability: its name is `saxon:for-each-group()` **[Saxonica, 2004]**. As with `analyze-string`, it is slightly clumsier to use than the custom syntax provided in XSLT 2.0, but again shows how much more power there would be in the language if higher-order functions were a standard feature.

2.4 Schema-aware processing

The most radical difference between XSLT 2.0 and XSLT 1.0 is that the language has become strongly typed, with a type system based on XML Schema. This has been done in such a way that untyped (schemaless) processing is still possible as a fallback. There are many reasons this change has taken place, and much debate about the desirability of making such a radical change, especially in view of the fact that XML Schema is widely criticized both for its complexity and for the limitations in its capability. I would like to concentrate here, however, on its impact for writing up-conversion applications.

Since up-conversion often starts with an input file that is not XML, it is unlikely that an XML Schema will exist to describe its structure. Fortunately this is not a problem: XSLT is still perfectly happy to work with untyped, schemaless data.

I have often found that it is best to structure an up-conversion as a sequence of two (maybe more) transformations. The first transformation takes the raw input data in whatever legacy format it arrives in, and translates it to an XML representation that is as close to the original structure as possible, consistent with it being XML. The second transformation takes this raw XML and translates it to the desired target XML vocabulary.

The target vocabulary typically represents XML that is designed to have significant visibility: it may be

long-lived, widely-shared, or both. Therefore, it is very likely that there will exist an XML Schema for this vocabulary. The schema-aware capabilities of XSLT that are relevant to up-conversion therefore tend to be those that are concerned with validating the result tree, rather than those concerned with processing the source. In the case study I will show how this validation assisted with the development process for creating correct XSLT transformations. The case study in this paper is an artificial one, it was constructed largely for pedagogic purposes, but I have had the same experiences in a real project involving the capture of human resources data from Excel spreadsheets for transfer into an XML database.

3. An up-conversion case study: GEDCOM

In this second part of this paper we will look at how the constructs introduced in the previous section are used in a practical example of an up-conversion exercise.

3.1 Description of the Problem

Genealogical data is interesting for a number of reasons. Genealogy is one of the most popular applications of the web for millions of people, and its success relies on the ability to exchange data between different application packages. The data itself is quite complex, for two reasons: the variety of information that people want to record, and the need to capture uncertain information and conflicting versions of events. For many years genealogical data has been exchanged using a format called GEDCOM **[LDS, 1996]**, devised by the Church of Jesus Christ of Latter-Day Saints (the Mormons). GEDCOM 5.5 uses a hierarchic record format rather in the style of a COBOL data definition, typified by the following entry:

```
0 @I53@ INDI
1 NAME Michael Howard /KAY/
1 SEX M
1 BIRT
2 DATE 11 OCT 1951
2 PLAC Hannover, Germany
3 MAP
4 LATI N52
4 LONG E9
1 OCCU Software Designer
2 DATE FROM 1975 TO 2004
1 EDUC Postgraduate
2 DATE FROM 1969 TO 1975
2 PLAC Cambridge, England
3 MAP
4 LATI N52
4 LONG E0
2 NOTE PhD in Computer Science
1 FAMS @F233@
1 FAMC @F221@
```

The `@I53@` field is a record identifier, and the values `@F233@` and `@F221@` are pointers to other records (specifically, the record describing the family in which this individual is a parent, and the record describing

the family in which this individual is a child).

This can of course be directly translated to an XML syntax, such as this:

```
<INDI>
  <NAME>Michael Howard /KAY/</NAME>
  <SEX>M</SEX>
  <BIRT>
    <DATE>11 OCT 1951</DATE>
    <PLAC>Hannover, Germany
      <MAP>
        <LATI>N52</LATI>
        <LONG>E9</LONG>
      </MAP>
    </PLAC>
  </BIRT>
  <OCCU>Software Designer
    <DATE>FROM 1975 TO 2004</DATE>
  </OCCU>
  <EDUC>Postgraduate
    <DATE>FROM 1969 TO 1975</DATE>
    <PLAC>Cambridge, England
      <MAP>
        <LATI>N52</LATI>
        <LONG>E0</LONG>
      </MAP>
    </PLAC>
    <NOTE>PhD in Computer Science</NOTE>
  </EDUC>
  <FAMS REF="F233"/>
  <FAMC REF="F221"/>
</INDI>
```

The first stage of our up-conversion application will be to convert the data into this form. After that we will see how to convert it further to the actual target XML vocabulary defined by the proposed GEDCOM-XML standard.

3.2 Stage One: Conversion to Raw XML

In my book *XSLT Programmer's Reference* (including the latest edition for XSLT 2.0) I describe how to perform this step by writing a GEDCOM parser in Java. The fact is, however, that it can be coded entirely in XSLT 2.0, and that the XSLT 2.0 code is actually shorter than the Java implementation. Let's see what it looks like.

First we have to read the input file, which we can do like this:

```
<xsl:param name="input" as="xs:string" required="yes"/>

<xsl:variable name="input-text"
              as="xs:string"
              select="unparsed-text($input, 'iso-8859-1')"/>
```

(I've actually cheated here. GEDCOM requires files to be encoded in a character set called ANSEL, otherwise ANSI Z39.47-1985, which is used for almost no other purpose. If ANSEL were a mainstream character encoding, it could be specified in the second argument of the `unparsed-text()` function call. In practice, however, it is rather unlikely that any XSLT 2.0 processor would support this encoding natively. Therefore, the conversion from ANSEL to a mainstream character encoding will still have to be done in a pre-processing phase.)

The next stage is to split the input into lines, which can be done using the XPath 2.0 `tokenize()` function. Since the `unparsed-text()` function does not normalize line endings (this might yet change) the regular expression for matching the separator between tokens accepts both UNIX and Windows line endings. The result is a sequence of strings, one for each line of the input file:

```
<xsl:variable name="lines"
              as="xs:string*"
              select="tokenize($input-text, '\r?\n')"/>
```

Now we need to parse the individual lines. Each line in a GEDCOM file has up to five fields: a level number, an identifier, a tag, a cross-reference, and a value. We will create an XML `line` element representing the contents of the line, using attributes to represent each of these five components:

```
<xsl:variable name="parsed-lines
              as="element(line)*">
  <xsl:for-each select="$lines">
    <xsl:analyze-string select="." flags="x"
                        regex="^([0-9]+)\s*
                               (@([A-Za-z0-9]+)@)?\s*
                               ([A-Za-z]*)?\s*
                               (@([A-Za-z0-9]+)@)?
                               (.*)$">
      <xsl:matching-substring>
        <line level="{regex-group(1)}"
              ID="{regex-group(3)}"
              tag="{regex-group(4)}"
              REF="{regex-group(6)}"
              text="{regex-group(7)}"/>
      </xsl:matching-substring>
      <xsl:non-matching-substring>
        <xsl:message>
          Non-matching line "<xsl:value-of select="."/>"
        </xsl:message>
      </xsl:non-matching-substring>
    </xsl:analyze-string>
  </xsl:for-each>
</xsl:variable>
```

Note first the `as` attribute on the `xsl:variable` declaration. I have consistently been declaring the types of my variables: this helps to pick up programming errors and it documents the stylesheet for the reader. I can do this even with a non-schema-aware stylesheet: the form `element(line)*` indicates that the variable holds a sequence of elements whose name is `line`. I could further constrain them to conform to a `line` element declaration in an XML schema by writing `schema-element(line)*`, but I've chosen not to do that here, because it's too much effort to create a schema to describe this transient data structure.

The actual content of the elements is constructed by analyzing the text of the input GEDCOM line using a regular expression. The attribute `flags="x"` allows the regex to be split into multiple lines for readability. The five lines of the regex correspond to the five fields that may be present. I describe this usage of `xsl:analyze-string` as a "single-match" usage, because the idea is that the regular expression matches the entire input string exactly once, and the `xsl:non-matching-substring` instruction is used only to catch errors. Within the `xsl:matching-substring` instruction, the content of the line is picked apart using the `regex-group()` function, which returns the part of the matching substring that matched the n'th parenthesized subexpression within the regex. If the relevant part of the regex wasn't matched (for example, if the optional identifier was absent) then this returns a zero-length string, and our XSLT code then creates a zero-length attribute.

So we now have a sequence of XML elements each representing one line of the GEDCOM file, each containing attributes to represent the contents of the five fields in the input. The next stage is to convert this flat sequence into a hierarchy, in which level 2 lines (for example) turn into XML elements that contain the corresponding level 3 lines.

Any problem that involves adding hierarchic levels to the result tree, that were not present in the source tree, can be regarded as a grouping problem, and it should therefore be no surprise that we tackle it using the `xsl:for-each-group` instruction. This time a group consists of a level N element together with the following elements up to the next one at level N. So this is a positional grouping rather than a value-based grouping. The option that we use to tackle this is the `group-starting-with` attribute, whose value is a match pattern that is used to recognize the first element in each group.

A single application of `xsl:for-each-group` creates one extra level in the result tree. In this example, we have a variable number of levels, so we want to apply the instruction a variable number of times. First we group the overall sequence of `line` elements so that each level 0 line starts a new group. Within this group, we perform a further grouping so that each level 1 line starts a new group, and so on up to the maximum depth of the hierarchy. As one might expect, the process is recursive: we write a recursive template that performs the grouping at level N, and that calls itself to perform the level N+1 grouping. This is what it looks like:

```
<xsl:template name="process-level">
  <xsl:param name="population" required="yes" as="element()*"/>
  <xsl:param name="level" required="yes" as="xs:integer"/>
  <xsl:for-each-group select="$population"
       group-starting-with="*[xs:integer(@level) eq $level]">
    <xsl:element name="{@tag}">
      <xsl:copy-of select="@ID[string(.)], @REF[string(.)]"/>
      <xsl:value-of select="normalize-space(@text)"/>
      <xsl:call-template name="process-level">
        <xsl:with-param name="population"
                        select="current-group()[position() != 1]"/>
        <xsl:with-param name="level"
                        select="$level + 1"/>
      </xsl:call-template>
    </xsl:element>
  </xsl:for-each-group>
</xsl:template>
```

When this is called to process all the `line` elements with the `$level` parameter set to zero, it forms one group for each line having the attribute `level="0"`, containing that line and all the following lines up to the next one with `level="0"`. It then processes each of these groups by creating an element to represent the level 0 line (the name of this element is taken from the GEDCOM tag, and its ID and IDREF attributes are copied unless they are empty), and constructs the content of this new element by means of a recursive call, processing all elements in the group except the first, and looking this time for level 1 lines as the ones that start a new group. The process continues until there are no lines at the next level (the `for-each-group` instruction does nothing if the population to be grouped is empty).

The remaining code in the stylesheet simply invokes this recursive template to process all the lines at level 0:

```
<xsl:template name="main">
  <xsl:call-template name="process-level">
    <xsl:with-param name="population"
                    select="$parsed-lines/ged/line"/>
    <xsl:with-param name="level"
                    select="0"/>
  </xsl:call-template>
</xsl:template>
```

This `main` template represents the entry point to the stylesheet. There is no `match="/"` template rule, because there is no source XML document with a root node to be matched; instead, XSLT 2.0 allows a transformation to be invoked by specifying the name of a named template where execution is to start. I use the name `main` as a matter of convention.

We have now converted the GEDCOM data to XML. The next step is to convert it to the actual XML vocabulary that the target application requires.

3.3 Stage Two: Conversion to the Target Schema

Like many up-conversion problems, the GEDCOM problem is best solved in two stages: the first stage is essentially a syntactic transformation of the raw data into XML, and the second stage is a semantic transformation to a different data model.

At the same time as moving to XML, the GEDCOM designers decided it was time to fix some long-standard deficiencies in the data model. The draft GEDCOM 6.0 specification **[LDS, 2002]** therefore not only moves from ANSEL character encoding to Unicode, and from COBOL-like level numbers to nested XML tags, it also changes the structure of the data. Events, for example, are now primary objects in their own right, rather than being always subsidiary to an individual or family. This reflects the fact that there is often uncertainty as to whether two events involve the same individual (rather than two distinct individuals having the same name), and it also makes it easier to record all the individuals associated with an event - for example, the witnesses at a marriage, or the godparents at a christening.

The transformation of GEDCOM 5.5 files to "raw XML", as described in the previous section, is therefore followed by a second transformation, this time to XML that conforms to the target schema defined by

GEDCOM 6.0. (I'm taking it as read here that GEDCOM 6.0 exists and is stable and is worth adopting as a target. This idealizes the actual state of affairs, but the debate isn't relevant to this paper.)

Multi-phase transformations can be done in either of two ways: using a single stylesheet (typically using different modes for the two phases) or using one stylesheet for each phase. I usually find it is easier to develop them using multiple stylesheets, and then integrate them together later as a production application.

The second transformation is rather more conventional than the first, because it starts with XML as its input. I've presented the full stylesheet in *XSLT 2.0 Programmer's Reference*, and I won't repeat it here in full. What I would like to draw out, however, is the impact of making this stylesheet schema-aware.

The first stylesheet, presented in the previous section, didn't use an XML schema. The input isn't XML, so it clearly has no schema; and the output uses a local transient XML vocabulary where the effort of writing a schema probably isn't worthwhile. However, for the second stylesheet, the aim is to produce output that conforms to a recognized standard XML vocabulary, for which an XML Schema exists, and we clearly want to have as much confidence as we can that the stylesheet output will always conform to this target schema.

With XSLT 1.0, the way you achieve this is to run your stylesheet against as many test cases as you can, and validate the output of each test case against the target schema. If validation errors are reported, you then have to debug the stylesheet to find out why it produced incorrect output in this particular case.

It would be far better if one could determine statically, purely from examination of the stylesheet, that its output will be correct. In practice this is unlikely to be fully achievable, because of the highly dynamic nature of XSLT template rules. However, there are many errors that could in principle be detected statically, and each error that is found this way makes a significant contribution to easing the testing and debugging burden. For example, here is an extract of the second-phase GEDCOM stylesheet:

```
<xsl:result-document validation="strict">
   <GEDCOM>
      <HeaderRec>
        <FileCreation Date="{format-date(current-date(),
                             '[D1] [MN,*-3] [Y0001]')}"/>
        <Submitter>
           <Link Target="ContactRec" Ref="Contact-Submitter"/>
        </Submitter>
      </HeaderRec>
      <xsl:call-template name="families"/>
      <xsl:call-template name="individuals"/>
      <xsl:call-template name="events"/>
      <ContactRec Id="Contact-Submitter">
        <Name><xsl:value-of select="$submitter"/></Name>
      </ContactRec>
   </GEDCOM>
</xsl:result-document>
```

One can see many potential errors that could be detected statically by the stylesheet compiler. It can check that there is a schema definition of the `GEDCOM` element, and that `HeaderRec` and `ContactRec` are permitted

respectively as the first and last child elements of the `GEDCOM` element. It can check similarly that the elements within the `HeaderRec` are allowed to appear where they do, that they are allowed to have the appropriate attributes, and that none of these elements have required attributes which the stylesheet does not generate. In some cases the compiler can also check that the textual content of elements and attributes is appropriate to their type. The analysis can extend beyond the fragment shown here to the three named templates invoked by this fragment; for example if the call on the `individuals` template preceded that on the `families` template, then the compiler could deduce that the stylesheet was outputting `IndividualRec` elements ahead of `FamilyRec` elements, which the schema does not allow.

As programmers, we are all familiar with the fact that errors detected at compile-time are much quicker to find and to fix than errors detected at run-time. This is as true for XSLT as for any other programming language.

Currently the only schema-aware XSLT processor available is my own Saxon product, and the current release (8.0) does not yet do the kind of static checking described above. Even run-time checking, however, can pay substantial dividends. For example, one error that I made during development was to write an attribute of a literal result element as `id="@ID"` instead of `id="{@ID}"`. Ordinarily, this would cause the result document to contain the attribute value `id="@ID"`. When the programmer gets round to validating the output (a stage which is often omitted during development and testing) this would reveal an error, because the `id` attribute is declared as having type `xs:ID`, and an `@` character is not allowed in values of this type. Running with a schema-aware processor, this error was reported as soon as the offending code in the stylesheet was executed, with the incorrect line in the stylesheet being accurately pinpointed.

I actually found that while developing this and other similar stylesheets, the number of errors detected by validation of result trees was so large that it became a little frustrating. Sometimes one actually wants to develop a stylesheet "top-down", getting the broad structure of the output right first, and focusing on the detail later. As a response to this experience, Saxon 8.1 allows multiple validation errors in the output to be reported in a single run, and it allows you to see the (invalid) result tree that was generated, along with comments inserted into the XML showing where it is invalid and which stylesheet instructions need to be changed to fix the errors. This provides another of the benefits normally associated with compile-time errors, the ability to report many errors in a single run.

Like other new features in XSLT 2.0, such as `xsl:analyze-string` and `xsl:for-each-group`, the facility to validate result documents on-the-fly is useful for a wide range of applications, of which up-conversion applications are just one example. But taken together, these features make a dramatic difference to the ease of developing up-conversion applications when compared with XSLT 1.0.

4. Conclusions

The first part of this paper described four specific features of XSLT 2.0 that make it highly suitable for writing up-conversion applications, namely:

- The unparsed-text() function
- Regular expression processing
- Grouping facilities
- Schema-aware processing

The second half of the paper showed how these features can be used in a practical up-conversion exercise, the translation of GEDCOM 5.5 genealogical data to the proposed GEDCOM 6.0 XML vocabulary.

XSLT 1.0 has been widely deployed to achieve both XML-to-XML and XML-to-text transformations. The conclusion of this paper is that XSLT 2.0 is also highly suited to a wide range of text-to-XML applications, thus greatly increasing the scope of applicability of the language.

Bibliography

[XSLT 2.0]
XSL Transformatons (XSLT) Version 2.0. W3C Working Draft 12 November 2003.

[XPath 2.0]
XML Path Language (XPath) 2.0. W3C Working Draft 23 July 2004.

[Kay, 2004a]
Michael Kay, XSLT 2.0 Programmer's Reference, 3rd edition. Michael Kay. Wiley, 2004

[Kay, 2004b]
Michael Kay, XPath 2.0 Programmer's Reference. Michael Kay. Wiley, 2004

[Tennison]
Jeni's XSLT Pages: Grouping. Jeni Tennison.

[LDS, 1996]
The GEDCOM Standard Release 5.5. Family History Department, The Church of Jesus Christ of Latter-day Saints. January 2, 1996

[LDS, 2002]
GEDCOM XML Specification, Release 6.0, Beta Version. Family and Church History Department, The Church of Jesus Christ of Latter-day Saints. December 6, 2002

[Saxonica, 2004]
Saxon 8.1 Documentation. Go to www.saxonica.com, follow links to Documentation, then Extensions, then Extension Functions. Saxonica Limited, to be published